NPS52-83-011

NAVAL POSTGRADUATE SCHOOL

Monterey, California

CONCURRENCY AND SYNCHRONIZATION IN
THE INTEL iAPX-432 PROTOTYPE SYSTEMS
IMPLEMENTATION LANGUAGE

Bruce J. MacLennan

September 1983

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-83-011	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: Concurrency and Synchronization in the Intel iAPX-432 Prototype Systems Implementation Language	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Bruce J. MacLennan	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Chief of Naval Research, Arlington, VA 22217	12. REPORT DATE: September 1983
	13. NUMBER OF PAGES: 26
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Concurrency, Synchronization, Message-passing, Intel iAPX-432, Coroutines, Processes, Tasks, Data-flow, Extensible Languages, Systems Implementation Languages, Petri-nets, actors, communication, communicating sequential processes.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes the concurrent execution and synchronization facilities of a prototype systems implementation language for Intel's iAPX-432 microprocessor. Full exploitation of the 432's facilities places many demands on a language intended for systems implementation. This report describes the prototype language's support for the 432's dynamic, message-based model of concurrency.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Date Entered)*

CONCURRENCY AND SYNCHRONIZATION IN THE INTEL iAPX-432 PROTOTYPE SYSTEMS IMPLEMENTATION LANGUAGE

B. J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

1. Introduction

This report describes the concurrent execution and synchronization facilities of a prototype systems implementation language for Intel's iAPX-432 microprocessor. Intel has kindly declared this work non-proprietary, so its publication is now possible [Brown83].

Full exploitation of the 432's facilities places many demands on a language intended for systems implementation. This report, which is an extension of Section 5.2 of [PSIL78], describes the prototype language's support for the 432's dynamic, message-based model of concurrency. Although the discussion should be comprehensible to anyone familiar with a modern data abstraction language, it will be helpful to first read the companion report [MacL83], which describes the prototype language's goals along with its abstraction mechanism.

2. Background

The concurrency and synchronization facilities of the prototype language provide for the synchronous and asynchronous communication of procedural abstractions. These facilities are based on a Petri-net/data-flow model of computation. This will not be emphasized in the following discussion, however, and knowledge of these is not a prerequisite to understanding the following.

The computational state at a given time is taken to be made up of some number of *actors* and some number of *exchanges*, through which the actors communicate[1]. Each actor has some number of *input* exchanges and some number of *output* exchanges. One exchange can be both an input and an output to a single actor. An actor can be *dormant*, which means that its internal state is incapable of changing, or it can be *active*, if its internal state is changing or capable of changing. Actors communicate by sending *messages;* a message is any value or object sent from one actor to another. A dormant actor can become active when messages are placed in certain of its input exchanges. Just which exchanges, or combinations of exchanges, varies from actor to actor, and is part of the definition of each actor. Once an actor becomes active, it can remain in that state for an indeterminate length of time before it becomes dormant again. During its active period it may have placed messages in its output exchanges.

There are two types of actors. First, there are *procedures*, which are similar to procedures and coroutines in other languages. They have bodies which are composed of statements and expressions that are executed sequentially (or collaterally) in the usual way. The other type of actor is an *activity*, which is composed of other exchanges and actors. Thus it can be seen that actors are built recursively. Actors are composed of other actors down to the lowest level, where there are only procedures. Procedures and activities are discussed in Sections 4 and 5, respectively. Actors, exchanges and messages are all *objects* (as opposed to *values*), see [MacL83, part 2].

3. Exchanges

An exchange is a place that can contain one or more messages. There are three kinds of exchanges.

The first kind of exchange is the *site*, which has the following characteristics:

1. It holds at most one message. When it does not hold a message it is said to be *empty*.

1. The reader will note that our approach is similar to Hewitt's actor formalism [Hewitt73, Hewitt75].

2. It is destructively read, i.e. accessing its contained message removes that message from the site and leaves it empty. Attempting to remove a message from an empty site causes the accessing actor to become dormant until sometime when the site is no longer empty.

3. A message can be put in a site only when it is empty. Attempting to put a message into a non-empty site causes the putting actor to become dormant until some time when the site is empty again.

The second type of exchange is the *pile*, which has the following characteristics:

1. A pile can hold any number of messages. There is no order implied among the messages in a pile.

2. Reading a pile returns and removes one of the messages it holds. If it is empty then the accessing actor becomes dormant until some time when the pile is non-empty.

3. Writing a pile causes the message to be added to the messages it already holds. It is always possible to put messages into a pile.

The third type of exchange is the *queue*, which has the following characteristics:

1. A queue can hold any number of messages. The messages are strictly ordered, however. That is, they form a sequence.

2. Reading a queue returns and removes the next message, i.e. the first element of the sequence of messages. If the queue is empty, then the accessing actor becomes dormant until some time when the queue is non-empty.

3. Writing a queue causes the message to be placed at its end of the sequence of messages, i.e. to be appended after the last element of the sequence. It is always possible to put messages into a queue.

In all the above cases, removing a message from an exchange and putting a message

into an exchange are considered indivisible operations. Thus it is not possible for two actors to simultaneously remove the same message from an exchange.

Exchanges are given names by bindings. The bindings that name exchanges have the syntax:

$$\textit{exch-binding}: \quad \textit{id-list}: [\textit{type}] \begin{Bmatrix} \textbf{site} \\ \textbf{pile} \\ \textbf{queue} \end{Bmatrix} [= \textit{exp}]$$

An exchange binding associates each element of the *id-list* with a separate exchange. The type-expression specifies that the exchange can only hold messages of that type. The expression *exp* is an optional initial value for the exchange. The initial value must be appropriate for the kind of exchange: single messages for sites, sets of messages for piles and sequences of messages for queues. If the initial value is omitted, then the exchange is made empty. If the type is omitted then it is taken to be the type of the initial value unless that is also omitted, in which case it is taken to be the type **token**, which is described below.

Tokens are used when the number of messages in an exchange is important, but their content is irrelevant. Tokens can be thought of as small atomic objects which are always distinct from each other. Token piles serve a function similar to *counter variables* [Gerber77]. They have many synchronization applications, as will be seen in the examples later. Sets of *n* tokens can be generated by the operation '*n* **tokens**'. Some examples of exchange bindings follow:

answer: **Bool site**;

Available: **pile** = k **tokens**;

messages: **string queue**;

4. Procedures

The procedure-binding allows a procedure to be invoked with a prefix, postfix, or infix syntax. All of these are illustrated in the examples in Figure 2. The digit indicates

proc-binding: **proc** [*digit*] { [*formals*] *infix-id* [*formals'*] | *id exch-binding* } [→ { *formals''* | *exch-binding* }] *pbody*

pbody: { = *exp* | **is** *st** **end** [*id*] }

infix-id: { *identifier* | *symbol* }

Figure 1. Syntax of Proc-Bindings

```
1. proc fac(n:int) → (f:int) is
     if n=0 then 1 → f;
     else n*fac(n-1) → f; end if;
   end;

2. proc divmod(x:int, y:int) → (q:int, r:int) is
     entier(x/y) → q;
     x-q*y → r;
   end;

3. proc fac(n:int) is
     if n=0 then return 1;
     else return n*fac(n-1); end if;
   end;

4. proc divmod(x:int, y:int) is
     entier(x/y) → q;
     return (q, x-q*y);
   end;

5. proc fac(n:int) = (n=0 ⇒ 1 | n*fac(n-1));

6. proc (n:int)! = (n=0 ⇒ 1 | n*(n-1)! );

7. proc (n:int) perm (r:int) = n!/(n-r)!;

8. proc (n:int) comb (r:int) = (n perm r)/r!;
```

Figure 2. Examples of Proc-Bindings

the procedure's precedence, with '**proc** 9's being the most binding and '**proc** 0's the least binding. If the digit is omitted it is assumed to be '9' The specification of the input and output exchanges is discussed later. The *pbody*, which is the body of the procedure, is composed of either an expression or a statement list. The statements are executed sequentially or collaterally, as defined elsewhere in the report [PSIL78].

Although *pbody* can contain any kinds of the statements or operators, one class of each is relevant here, *viz.* the *communicators* (Figure 3).

communicator:	{**return** [*source*] / *transition* [**waiting** *transition*]}
transition:	{*source* [→ *destination*] / *destination* ← *source*}

Figure 3. Syntax of Communicators

The full description of *source* and *destination* is deferred to the discussion of activities. Briefly, a *source* can be described as anything that can provide a message, including an expression, exchange or record composer, and a *destination* can be described as anything that can accept a message, such as a variable, exchange, or record decomposer. Note that the simplest case of a transition is an assignment statement. Thus,

x → y; or

y ← x;

both take a message from exchange x and put it in exchange y. Of course, if x is empty, this statement will wait until x holds a value. The **waiting** option on transitions is discussed in Section 6.

Since messages frequently take the form of records, it is quite common to have a *record composer* as the source in a transition statement. For example, if we have the declarations

var m, n: **int**;

message = **record** x: **int**; y: **int**; **end record**;

port: message **site**;

then the transition

(m,n) → port;

will take the messages in variables 'm' and 'n', and compose them into a record, which is then placed in the exchange called 'port'.

Conversely, it is common to have a *record decomposer* as the destination of a communicator:

port → (m,n);

This statement waits until there is a message in the exchange called 'port', at which time it takes the message and assigns its components to the variables called 'm' and 'n'.

If we use a composer and a decomposer in the same transition, then we have the effect of a simultaneous assignment:

(m,n) → (n,m);

This composes m and n into a record, whose components are immediately decomposed and assigned to n and m. Thus, we have exchanged the contents of m and n. A more complex example is:

(m-n, m-n) → (m,n);

It would be difficult to write this without the simultaneous assignment effect of the transition statement.

Suppose that the input exchange of a procedure 'DivMod' is called 'in' and that the output exchange is called 'out':

```
proc DivMod in: (record x:int; y:int; end) site
        → out: (record q:int; y:int; end) site is
    ...
    end DivMod;
```

As is often the case, the input and output messages to DivMod are records. We can send a message to DivMod's input exchange, thus initiating the execution of DivMod, by the transition:

(m,n) → DivMod.in;

We have used the record composer (m,n) to form a pair of the integers m and n; this record is then placed in DivMod's input exchange (DivMod.in).

In an exactly analogous way we can accept a value from DivMod's output exchange (DivMod.out). The transition

DivMod.out → (j,k);

will wait until there is a message in DivMod's output exchange. This message will be taken from the exchange and be broken down by the record decomposer (j,k), i.e., its components will be placed in the variables j and k.

We consider the special case of *synchronous communication*. It is frequently the case that when a message is sent to an actor, computation in the sender cannot proceed until an answer is received from that actor. The send is followed by an immediate wait. For example, we would send a message to DivMod and immediately wait for a response by

(m,n) → DivMod.in;

DivMod.out → (j,k);

We allow this to be abbreviated by the transition statement

DivMod(m,n) → (j,k);

This can be read: "Send (m,n) to DivMod and wait for a reply, which is to be put into (j,k)."

In general, a transition statement such as

$f\ e \rightarrow d$;

is an abbreviation for the pair of transitions

$e \rightarrow f$.in;

f.out → d;

Whenever control enters $f\ e \rightarrow d$, the message e is put into the input exchange of f,

and the calling procedure becomes dormant until *f* answers, whereupon the answer is placed in *d*. In an expression context, such a synchronous communication can be written '*f e*'. For example,

```
fac(n) → d;
put(d) → e;
```

can be written

```
put( fac(n) ) → d;
```

Indeed, an expression is just a nesting of synchronous communications. For example,

```
q ← fac( comb( fix(m), n) );
```

is just an abbreviation for

```
fix(m)      → t0;
comb(t0,n)  → t1;
fac(t1)     → q;
```

which is in turn an abbreviation for

```
(m)         → fix.in;
fix.out     → t0;
(t0,n)      → comb.in;
comb.out    → t1;
(t1)        → fac.in;
fac.out     → q;
```

The input/output exchanges of a procedure can take two forms. In the simplest case they are just exchange bindings. This defines the identifier by which the exchange is known within the procedure and specifies the exchange's type. This can be seen in the previous definition of 'DivMod'. The second form is described below.

The object passed to procedures are usually *n*-tuples of values, i.e. records. This is

certainly the case for prefix function of more than one argument and, for uniformity, is also taken to be the case for other procedures. Because input/output exchanges are usually of record types, a special abbreviation is provided, a *formals*, which is essentially a record type definition. For example, the procedure declaration

```
proc DivMod in: (record x:int; y:int; end) site
        → out: (record q:int; r:int; end) site is
  let var x, y, q, r: int;
  in → (x,y);
  entier(x/y) → q;
  x-q*y → r;
  (q,r) → out;
  end DivMod;
```

can be abbreviated as shown in example 2 in Figure 2.

The semantics of this style of input/output exchange is as follows: there is an anonymous input exchange of the record-type. Whenever a message arrives at this exchange it is immediately broken down into its components, which are assigned to the variables in the *formals*. Similarly, whenever the procedure exits (which process is described below), the values of the names in *formals*' are gathered together and composed into a record which is sent to the output exchange of the procedure.

The most common way of specifying the input/output exchanges is the record-type abbreviation. The case where the inputs or outputs are not *formals*, while more primitive, is less common. It is only used when an entire parameter package must be manipulated as a unit.

Since a dormant procedure will never become active if it never receives an input, if both *formals* and *formals*' are omitted, then the input exchange is assumed to be '()', a site of type **token**. If two inputs are specified, then they must both be present before the procedure will become active. This follows from the semantics of record

composers: all the records components must be available before the record can be built.

Note that the operation 'in → *dest*' causes a procedure to become dormant until there is a value in its input exchange. When such a value arrives the procedure may become active, and when it does become active that input will be put in *dest*. At the beginning of every procedure there is an implicit operation of this form that accepts the procedure's parameters.

The other kind of communicator is '**return** *source*'. It is equivalent to '*source* → out' followed by a transfer back to the implicit 'in → *dest*' at the beginning of the procedure. This is the usual mechanism for returning results from a serially reusable procedure; see example 4 in Figure 2. There is an implicit **return** at the end of every procedure.

Since the **return** statement references the output exchange anonymously, the declaration of this exchange can be omitted from the procedure binding. See examples 2 and 3 (Figure 2) and compare to examples 1 and 2.

In practice many procedures are composed of a single **return** statement:

proc *x f x'* **is**

return *s*;

end *f*;

where *s* is a source. These procedures can be abbreviated as

proc *x f x'* = *s*;

The output specification (*formals''*) can be included if it aids readability or forces a coercion. See examples 5, 6, 7 and 8 (Figure 2) for this type of procedure binding. Particularly compare to examples 1, 3, 5 and 6.

activity−binding:	**activity** [*digit*] [*fxl*] *infix−id* [*fxl'*] [→ *fxl''*] is *abody* end
abody:	{*binding* / *transition*}, ...
fxl:	[*formal exchange list*]

Figure 4. Syntax of Activities

5. Activities

As indicated previously, an *activity* is an abstraction mechanism whereby actors are combined to form larger actors. Similar to the *formals* of a procedure are the formal exchange lists, *fxl*, *fxl'* and *fxl''*, of an activity. Each of these is composed of an assemblege of exchanges. There is, however, a significant difference between a formal parameter of a procedure and a formal exchange of an activity. The formals of a procedure are "synchronized," i.e. all inputs must be ready before the procedure can become active. The inputs of an activity are not synchronized in this way. In other words, an activity can become active as soon as there are messages in such inputs as will activate one or more of the activity's subactors. Other inputs may arrive while the activity is active. Thus there may be more than one locus of control in an activity at a time. In a very real sense, an activity is just an abstraction of part of a data-flow network.

An activity body (*abody*) is composed of bindings and transitions. Normally the bindings will associate identifiers with exchanges and other actors. The transitions describe the connections between formal exchanges, local exchanges, non-local exchanges and other actors. In contrast to a procedure body, there is no sequential flow of control through an activity body. Within the body of an activity, transitions execute when they are ready to execute.

For an example, we define an activity 'Merge' which nondeterministically merges the contents of two input queues into an output queue. The body of the activity is two simple transitions:

activity Merge [q,r: message **queue**] → [s: message **queue**] **is**

q → s,

r → s

end;

Whenever q is not empty the first transition can fire and move a message from the beginning of q to the end of s. Also, whenever r is not empty the second transition can fire and move a message from r to s.

6. Transitions

6.1 Communication Primitives

Transitions can occur in two contexts: as statements and as declarations. The meaning of a transition *statement*, as discussed previously, is to move a value from the source(s), through an actor, to the destination(s). The actors themselves can either be built-in operators, or user-defined actors (procedures or activities), or actor-variables. (The latter are analogous to procedure variables in other languages.)

Transition *declarations* have the same syntax as transition statements. The difference is that they do not order any action to take place; they merely define the connections among a set of actors and exchanges. We have already seen transition declarations in the body of an activity.

At the lowest level in the expression syntax are primaries, which can come in several forms. We have already seen the *composer*, which is the mechanism used to construct a record from its components. As such, it also constitutes an actual parameter list to a procedure. The parentheses of the composer remind one of the parentheses in the formals of the procedure binding. The actual parameters to the composer are made to correspond to the position dependent and position independent fields of the record-type, in a manner described in [PSIL78].

Actual parameters are passed to activities in an analogous manner, except that

square brackets are used. The bracketed list of (position dependent and position independent) parameters is supposed to remind one of the bracketed formal exchange lists in the activity binding. For example, if Source1, Source2, and Sink are three queues of type message:

Source1, Source2, Sink: message **queue**;

then we can use the Merge activity to merge Source1 and Source2 into Sink by writing the transition declaration:

Merge [Source1, Source2] → Sink

More precisely, the text of Merge is *instantiated* with 'Source1' and 'Source2' substituted for the formal input exchanges ('q' and 'r') and 'Sink' substituted for the formal output exchange 's'. Notice that brackets around 'Sink' are not required since there is only one output from Merge.

The *sequencer* is a special form of the identity operation and is represented by a sequence of sources separated by semicolons. All these sources must be available before the sequencer becomes active. When it does become active, all its inputs are accepted, and its value is the value of the last source in the sequence of sources. Thus, when x, y and z are all non-empty, the value of z is placed in p by:

[x;y;z] → p;

Thus, a sequencer can be thought of as a gate: when x and y are present it gates z into p. Typical applications of sequencers can be found in the synchronization examples, later.

The expression '**empty** *primary*' is called an *inhibitor*, and is used for testing the emptiness of exchanges. If *ex* is empty then '**empty** *ex*' will produce one token as its value when a value is requested. If *ex* is not empty, then '**empty** *ex*' is empty. Thus,

[**empty** x; y] → z;

will move a value from y to z only if x is empty. Inhibitors are frequently used with

sequencers, as this example indicates.

There are a number of ways to handle the distribution of values returned by a procedure or activity. They are described in the following paragraphs.

A destination determines the disposition of the contents of a single output exchange. In the simplest case a destination is just a primary that refers to an exchange or variable. In this case the value is placed in that exchange or variable. Thus, if e is an exchange,

f[x,y] → e;

will place the result of f[x,y] into e. Often it is desirable to place a value in several different exchanges. This is done with a *distributer:*

f[x,y] → {d,e};

Analogous to synchronization of inputs is *desynchronization* of outputs. As explained above,

[x;y;z] → p;

moves a value from z to p only if the "control values" x and y are present. In an exactly analogous way,

p → [x;y;z];

moves p to z and generates "control values" (i.e. tokens) in x and y. Example applications are found in the synchronization examples.

As discussed earlier, a *decomposer* performs the opposite operation of a *composer*. Thus, if z is a complex number (with two position independent real fields Re and Im), its real and imaginary components can be assigned to x and y, respectively, by:

z → (Re:x, Im:y);

The **waiting** option is discussed in the next section.

6.2 Non-Hierarchical Synchronous Communication

The **waiting** option on transitions provides a mechanism for non-hierarchical synchronous communication (e.g. "coroutine" communication). If *f* and *p* are exchanges, *s* is a source and *d* is a destination, then

$$s \rightarrow f \ \textbf{waiting}\ p \rightarrow d;$$

means the same as

$$s \rightarrow f; p \rightarrow d;$$

That is,'*s* → *f* **waiting** *p* → *d*;' means 'send source *s* to exchange *f* and wait for a response in exchange *p*, which is to be placed in dest *d*'. If the response from *p* is not needed (i.e., it is only for synchronization), then '→*d*' can be omitted. Examples are given in the discussion of coroutines, below.

Non-hierarchical synchronous communication ("coroutine" communication) denotes the process whereby several procedure-like objects communicate without a definite caller/callee relationship. The facilities necessary to communicate in this way have already been introduced. How they are used will be illustrated by example.

The application is a text-justifier[2]. It will read lines of characters off an input file and write them on an output file with blanks inserted between words so that all lines are the same length. The program will be organized as a pipeline, with four *form objects* [MacL83] comprising the pipe:

1. CharReader - reads characters from the input file, ignoring end-of-lines.
2. WordReader - divides the character stream into words, ignoring repeated blanks.
3. Justifier - generates strings of blanks to separate the words so that their total length is as required, and generates an end-of-line character at the end of each line.

2. This is also the example used in [Dahl72], thus permitting comparison of the coroutine mechanisms in Simula and the prototype language.

```
StringWriter = obj form
 public inp: string site;

 public proc start is
  let var s: string;
  repeat
   () → Justifier.out waiting inp → s;
   for ch in s repeat outfile.put(ch); end;
  until s = string(eof);
  end start;
end StringWriter;
```

Figure 5. Coroutine Example: StringWriter

```
Justifier = obj form
 public inp: string site;
 public out: site;
 ...
 public proc init → answer: site is
  let var s, t: string;
  () → answer waiting out;
  Repeat
   () → WordReader.out waiting inp → s;
   ...
   t → StringWriter.inp waiting out;
   ...
   end repeat;
  end init;
 end Justifier;
```

Figure 6. Coroutine Example: Justifier

4. StringWriter - writes the generated character strings to the output file.

The pipe is set up and initiated by the following instructions:

```
CharReader.init;

WordReader.init;

Justifier.init;

StringWriter.start;
```

The first three invocations allow the 'init' procedures in CharReader, WordReader and Justifier to initialize whatever they might have to. They then answer, leaving themselves ready to begin work. The last invocation, StringWriter.start, allows StringWriter to initialize itself, but rather than waiting, it goes directly to work by requesting a string from Justifier. The code for StringWriter is in Figure 5. StringWriter declares a public site 'inp', at which it will wait for strings to write. (In general, we will use the

```
WordReader = obj form
  public out: site;
  public inp: char site;
  var s = '';   % s is a string variable initalized to null string

  public proc init → answer: site is
    () → answer waiting out;
    Repeat
      let var ch: char;
      () → CharReader.out waiting inp → ch;
      if ch <> ' ' then s.append(ch);
      elseif s <> '' then
        s → Justifier.inp waiting out;
        s ← '';
      end if;
    end repeat;
  end init;
end WordReader;


CharReader = obj form
  public out: site;

  public proc init → answer: site is
    () → answer waiting out;
    until eof infile repeat
      (next infile) → WordReader.inp waiting out;
      end;
    eof → WordReader.inp waiting out;
    end init;
end CharReader;
```

Figure 7. Coroutine Example

identifier 'inp' for messages going from the start of the pipe toward the end, and 'out' for acknowledgements going from the end toward the start.) StringWriter immediately enters a loop. It sends a request for a string to Justifier (through Justifier.out) and then waits for an answer at 'inp'. When this string arrives, the characters in it are written out one at a time. If the string was **eof** (end of file), then StringWriter terminates by returning to its caller

The next element of the pipe is Justifier. Since it is fairly complicated, only the parts necessary for this discussion are shown in Figure 6. Justifier has two communication sites, 'inp' through which it waits for strings form WordReader, and 'out', at which it waits for requests for strings from StringWriter. After Justifier completes initialization, it exits to the procedure controlling the pipe by:

() → answer **waiting** out;

where 'answer' denotes the output exchange of the procedure. Thus, this transition means: send an answer back to the anonymous caller, and then wait at 'out' for a request from StringWriter. When the first request arrives, Justifier enters its main loop, iterating once for each line.

The body of the loop requests words from WordReader, as it needs them, by:

() → WordReader.out **waiting** inp → s;

The expression on the left of **waiting** sends the request to WordReader.out. Justifier then waits for a string at 'inp', which upon arrival is placed in 's'. The body of the loop sends strings to StringWriter, as they are ready, by:

t → StringWriter.inp **waiting** out;

The effect of this transition is to send t to the exchange StringWriter.inp, and to wait at 'out' for another request.

The bodies of the remaining two pipe elements are shown in Figure 7. They are not described, since they follow the same pattern.

```
PreciousResource = obj form
 Ready: site = 1 tokens;
 Busy: site;

 public proc Red is
  Ready → Busy;
  ... Red's critical section ...
  Busy → Ready;
  end Red;

 public proc Blue is
  Ready → Busy;
  ... Blue's critical section ...
  Busy → Ready;
  end Blue;

 end PreciousResource;
```

Figure 8. Mutual Exclusion Example

7. Synchronization

All the facilities necessary for synchronization have already been introduced. Their use will be indicated through a number of examples.

7.1 Mutual Exclusion

Two procedures, Red and Blue, share a resource. It is required that they do not execute concurrently. Synchronization is accomplished by having them share a site, Ready, which contains a token if and only if the resource is *not* being used. The solution is in Figure 8. Operation is as follows: Control enters Red unobstructed. Before it can enter its critical section, however, Ready must be present. When it is, control passes through the critical section, performing the Red operation. When this is completed, a token is sent back to Ready, thus releasing the resource. Blue operates analogously.

```
Buffer = obj form
  Mes: message site;

  public proc Deposit(m:message) is
    m → Mes;
    end Deposit;

  public proc Remove is
    return Mes;
    end Remove;

  end buffer;
```

Figure 9. Single-Slot Buffer Example

```
BufferManager = obj form
  Avail: pile = N tokens;
  Buffer: message queue;

  public proc Deposit(m:message) is
    [Avail; m] → Buffer;
    end Deposit;

  public proc Remove → (m:message) is
    Buffer → [Avail; m];
    end Remove;

  end BufferManager;
```

Figure 10. Multiple-Slot Buffer Example

7.2 Single-Slot Buffer

Two procedures, Deposit and Remove, share a message buffer, Mes. Deposit will be allowed to put a message in the buffer only when it is empty, and Remove will be allowed to read it only when there is a message there. The solution is in Figure 9. Only if Mes is empty will the transition in Deposit execute and copy m into Mes. Similarly, only if Mes is occupied will Remove be able to empty it and return the message.

7.3 Multiple-Slot Buffer

Two procedures, Deposit and Remove, share a queue, Buffer, that can hold N messages. Deposit can execute only if a slot in Buffer is available and Remove can execute only if a slot in Buffer contains a message. A pile Avail will contain a token for each available slot. The solution is in Figure 10. Deposit waits until there is a token in Avail. The transition then fires, placing the message in Buffer. When there is a message in Buffer it will be possible for Remove to execute, placing a token in Avail and returning the message.

7.4 Concurrent Readers

Two procedures, Read and Write, share a resource. No writing can take place when reading is in progress, but any number of readers can be active at one time. A site, Writing, will contain a token if a write is in progress and a pile, Reading, will contain a token for each Read in progress. A site, Ready, contains a token whenever the state of the other two exchanges is stable. (Such a mutual exclusion site is usually required when inhibitory inputs are used. Inhibitory inputs, by their nature, are not self-synchronizing.) The solution is in Figure 11. The actions of Read are as follows: The Read waits until there is a token in in Ready and there is no writing in progress. It then indicates that it is Reading. When reading has been completed, a token is removed from Reading. The action of Write is analogous.

```
PreciousResource = obj form
  Ready: site = 1 tokens;
  Writing: site;
  Reading: pile;

  public proc Read is
    [Ready; empty Writing] → [Ready; Reading];
    ... perform read operation ...
    Reading → {};
    end Read;

  public proc Write is
    [Ready; empty Writing; empty Reading] → [Ready; Writing];
    ... perform write operation ...
    Writing → {};
    end Write;

end PreciousResource;
```

Figure 11. Concurrent Readers Example

```
PreciousResource = obj form
  Ready: site = 1 tokens;
  Writing: site;
  WriteRequested, Reading: pile;

  public proc Read is
    [Ready; empty WriteRequested] → [Ready; Reading];
    ... perform read operation ...
    Reading → {};
    end Read;

  public proc Write is
    1 tokens → WriteRequested;
    [Ready; empty Reading; empty Writing] → [Ready; Writing];
    ... perform write operation ...
    [Writing; WriteRequested] → {};
    end Write;

end PreciousResource;
```

Figure 12. Write-Priority Example

7.5 Concurrent Readers with Write Priority

The previous solution has a problem, namely that a continuous stream of Read requests can block Writes forever. This can be solved if we stipulate that no new Reads can begin if a Write is trying to get access to the resource. Further, we can require that any Write requests arriving when a Write is already in progress will be serviced before any new Reads are allowed to begin. The solution is similar to the previous. The

only change necessary is to introduce a new pile, WriteRequested, which contains one token for each Write attempting access to the resource. The solution is in Figure 12. The only difference from the operation of the previous example is the following. Whenever Write begins executing, it immediately sends a token to WriteRequested, thus blocking any further Reads. This token is removed when the Write operation is completed.

```
activity Spool [PrintQ: listing queue, CommandQ: string queue,
                PStatus: status site] → [PCommand: IOcommand site] is
  Ready: site = 1 tokens,
  Done, Restart: site,
  Start, Hold: listing site,

  [Ready; PrintQ] → {Start, Hold},
  Driver [Start, CommandQ, Pstatus] → [Done, Restart, PCommand],
  [Hold; Done] → Ready,
  [Restart; Hold] → {Start, Hold}
end Spool;
```

Figure 13. Printer Spooler Example

```
activity PrinterManager [PrintQ: listing queue,
  PCmd: (string queue) array {1..3} ] is

  Spool [PrintQ, PCmd[1], Lpr[1].out] → Lpr[1].in,
  Spool [PrintQ, PCmd[2], Lpr[2].out] → Lpr[2].in,
  Spool [PrintQ, PCmd[3], Lpr[3].out] → Lpr[3].in
end PrinterManager;
```

Figure 14. Printer Manager Example

7.6 Printer Spooler

In this section we define an activity 'Spool' that controls a printer. This activity will have three input exchanges and one output exchange. The input exchanges are:

1. A queue containing listings to be printed.
2. A queue containing commands from the operator or system console (for example, to restart the print job).
3. A site containing any status information returned by the printer controller.

The output exchange is a site through which I/O commands can be sent to the printer controller.

We assume that an activity called 'Driver' is available that has three formal input exchanges and three formal output exchanges. The input exchanges are:

1. A site containing a listing to be printed.

2. A queue containing commands from the operator.

3. A site containing status information returned by the printer controller.

The output exchanges are:

1. A site indicating that the driver has completed printing a listing.

2. A site indicating that the driver has aborted printing the listing and needs to restart it.

3. A site containing an I/O command to be sent to the printer controller.

The Spool activity should operate as follows: If the driver is ready and a listing is waiting to be printed, then that listing should be sent to the driver. However, the listing must also be saved in case a request to restart it is received. If the listing is completed normally, then this extra copy is discarded. If a restart request is received, then the driver must start over with it. The activity to implement these functions is shown in Figure 13.

Now suppose that we have three printers and that we need an instance of Spool to manage each. We will assume that 'Lpr' is the name of an array containing three records that contain the I/O control and status sites for these printer's controllers. We define an activity 'PrinterManager' that has the following input exchanges:

1. A queue containing listings to be printed.

2. An array containing a command queue for each spooler.

The definition is shown in Figure 14.

8. References

[Brown83] Brown, W. L. personal communication, March 11, 1983.

[Dahl72] Dahl, O. -J. Hierarchical Program Structures, in Dahl, O. -J.; Dijkstra, E. W.; and Hoare, C. A. R., *Structured Programming*, Academic Press, 1972, pp. 175-220.

[Gerber77] Gerber, A. J. Process Synchronization by Counter Variables, *SIGOPS Operating Systems Review 11*, 4, October 1977, pp. 6-17.

[Hewitt73] Hewitt, Carl; Bishop, Peter; and Steiger, Richard. A Universal Modular Actor Formalism for Artificial Intelligence, *International Joint Conf. on Artificial Intelligence - 73*, Stanford, CA, August 1973.

[Hewitt75] Hewitt, Carl. Protection and Synchronization in Actor Systems, *ACM SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication*, Santa Monica, CA, March 24-25, 1975.

[MacL83] MacLennan, B. J. Abstraction in the Intel iAPX-432 Prototype Systems Implementation Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-004, April 1983.

[PSIL78] Brown, W. L.; and MacLennan, B. J. *INTEL 8800 Prototype Systems Implementation Language Specification*, March 8, 1978 (revised August 2, 1978; January 24, 1979; December 21, 1979).

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Hq Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217	1
Dr. David Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	1
Professor John M. Wozencraft, 62wz Department of Electrical Engineering Naval Postgraduate School Monterey, CA 93940	1
Professor Uno Kodres, Department of Computer Science Naval Postgraduate School Monterey, CA 93940	1
Mr. William L. Brown Intel Corporation 5200 N.E. Elam Young Parkway Hillsboro, OR 97123	1

Mr. H. M. Gladney
IBM Research Laboratory
5600 Cottle Road
San Jose, CA 95193

1

Dr. Mary Vernon
Computer Science Department
3732 J Boelter Hall
University of California
Los Angeles, CA 90024

1